Structured Programming Language (CSE113)
Lecture # 09
Pointer in C programming

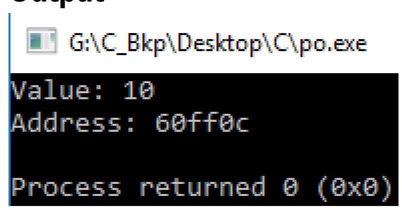
What is a pointer?
A **pointer** is a variable that stores the address of another variable. Pointers are used in C program to access the memory and manipulate the address.

Pointer Syntax:
data_type *pointer_variable_name;

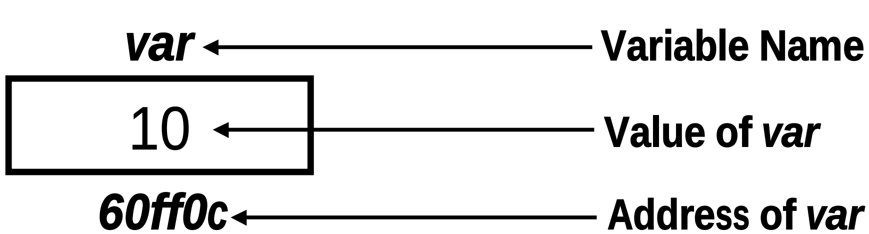
Example: int *p; char *p;

Where, * is used to denote that “p” is pointer variable and not a normal variable.

Address in C:
If you have a variable ***var*** in your program, **&var** will give you its address in the memory, where **&** is commonly called the reference operator. The **&** operator is also known as “**Address of**” Operator.

Source Code	**Output**
#include <stdio.h> int main() { int var = 10; printf("Value: %d\n", var); printf("Address: %x\n", &var); return 0; }	

Note: You may obtain different value of address while using this code.



Reference operator (&) and Dereference operator (*)

As discussed, **&** is called reference operator. It gives you the address of a variable.

Likewise, there is another operator that gets you the value from the address, it is called a dereference operator *****.

Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Source Code

```
#include <stdio.h>
int main()
{
  int* pc, c;

  c = 22;
  printf("Address of c: %x\n", &c);
  printf("Value of c: %d\n\n", c);

  pc = &c;
  printf("Address of pointer pc: %x\n", pc);
  printf("Content of pointer pc: %d\n\n", *pc);

  c = 11;
  printf("Address of pointer pc: %x\n", pc);
  printf("Content of pointer pc: %d\n\n", *pc);

  *pc = 2;
  printf("Address of c: %x\n", &c);
  printf("Value of c: %d\n\n", c);
  return 0;
}
```

Output

```
Address of c: 60ff08
Value of c: 22

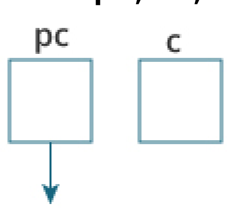
Address of pointer pc: 60ff08
Content of pointer pc: 22

Address of pointer pc: 60ff08
Content of pointer pc: 11

Address of c: 60ff08
Value of c: 2
```

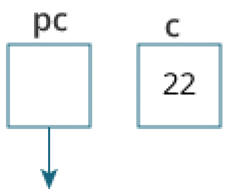
Explanation of the program

1. `int* pc, c;`



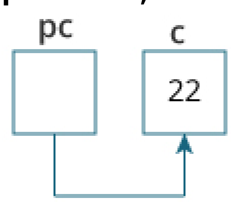
Here, a pointer ***pc*** and a normal variable ***c***, both of type **int**, is created.
Since ***pc*** and ***c*** are not initialized at first, pointer ***pc*** points to either no address or a random address. And, variable ***c*** has an address but contains a random garbage value.

2. **c = 22;**



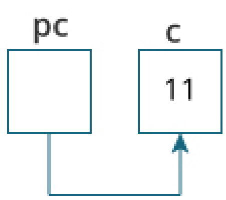
This assigns 22 to the variable *c*, i.e., 22 is stored in the memory location of variable *c*. Note that, when printing &c (address of c), we use %x rather than %d since address is usually expressed as a hexadecimal number. We can also use %u unsigned integer (always positive).

3. **pc = &c;**



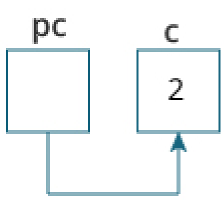
This assigns the address of variable *c* to the pointer *pc*.
You see the value of *pc* is same as the address of *c* and the content of *pc* is 22 as well.

4. **c = 11;**



This assigns 11 to variable *c*.
Since, pointer *pc* points to the same address as *c*, value pointed by pointer *pc* is 11 as well.

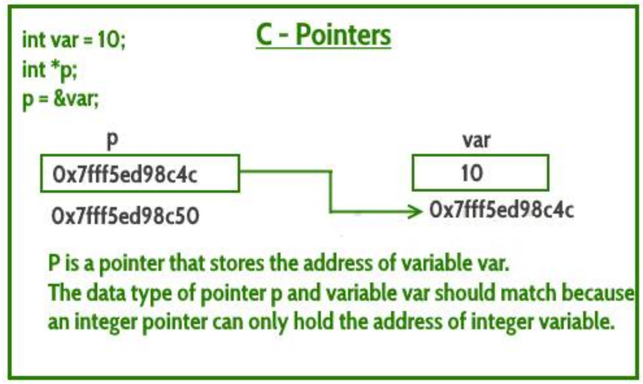
5. ***pc = 2;**



This change the value at the memory location pointed by pointer *pc* to 2.
Since the address of the pointer *pc* is same as the address of *c*, value of *c* is also changed to 2.

Another Example

Example of Pointer demonstrating the use of & and *

Source Code	Output
#include <stdio.h> int main() { int *p; int var = 10; p = &var; printf("Value of variable var is: %d\n", var); printf("Value of variable var is: %d\n", *p); printf("Address of variable var is: %p\n", &var); printf("Address of variable var is: %p\n", p); printf("Address of pointer p is: %p\n", &p); return 0; }	Value of variable var is: 10 Value of variable var is: 10 Address of variable var is: 0x7fff5ed98c4c Address of variable var is: 0x7fff5ed98c4c Address of pointer p is: 0x7fff5ed98c50 

Note: %p is used to get the address in hexadecimal number

Common mistakes when working with pointers

Suppose, you want pointer ***pc*** to point to the address of ***c***. Then,

int c, *pc;

// Wrong! pc is address whereas,
// c is not an address.
pc = c;

// Wrong! *pc is the value pointed by address whereas,
// &c is an address.
***pc = &c;**

// Correct! pc is an address and,
// &c is also an address.
pc = &c;

// Correct! *pc is the value pointed by address and,
// c is also a value (not address).
***pc = c;**

1. Write a C program to swap two numbers using pointers (Call by reference/call by address).

Source Code

```
#include<stdio.h>
void swap(int *a, int *b);
int main()
{
   int a = 20, b = 50;
   printf("Before Swapping: a = %d, b = %d\n",a,b);
      swap(&a, &b);
   printf("After Swapping: a = %d, b = %d\n",a,b);
   return 0;
}
void swap(int *a, int *b){
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Output

```
Before Swapping: a = 20, b = 50
After Swapping: a = 50, b = 20
```

Call by Value vs. Call by Reference

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. Call by value:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter cannot be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

2. Call by reference:

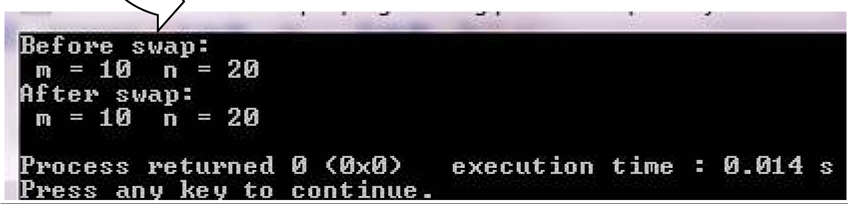
- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

Example of Call by Value:

```
#include<stdio.h>
void swap(int a, int b);
 int main(){
   int m = 22, n = 44;
   printf("Before swap:\n  m = %d\t  n = %d\n", m, n);
     swap(m, n);
   printf("After swap:\n  m = %d\t  n = %d\n", m, n);
}
 void swap(int a, int b) {
   int tmp;
      tmp = a;
      a = b;
      b = tmp;
}
```

Output shows that there are no changes in the values, though they had been changed inside the function.

Output:

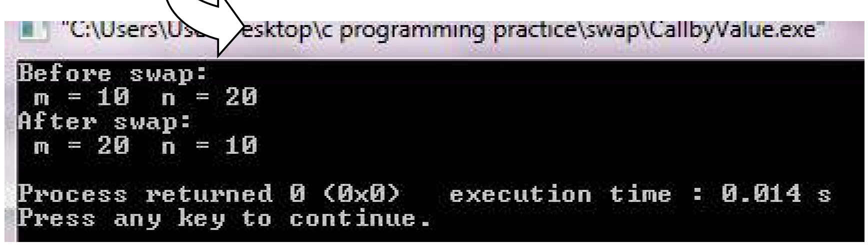


Example of Call by Reference:

```
#include<stdio.h>
void swap(int *a, int *b);
 int main(){
   int m = 22, n = 44;
   printf("Before swap:\n  m = %d\t n = %d\n", m, n);
     swap(&m, &n);
   printf("After swap:\n  m = %d\t n = %d\n", m, n);
}
 void swap(int *a, int *b) {
   int tmp;
     tmp = *a;
     *a = *b;
     *b = tmp;
}
```

Output shows that the change has reflected **outside the function** as well, unlike call by value where the changes do not reflect outside the function.

Output:



Thank You ☺